????-??
March 8, 2007

The Connection Between Hits and Geometry

Rob Kutschke, Fermilab CD

Abstract

This note presents a sketch of how one might connect tracking hit information with its associated geometry information, where "tracking" includes both pixels and strips. The note is intended to as a way point on the discussion we started a few weeks ago and will include: a sketch of how hit information flows, in experimental data, from the detector to its incorporation into tracks; the corresponding path for hits from Monte Carlo events; and a discussion of matching Monte Carlo reconstructed tracks with generated tracks.

Contents

1 Introduction

For track finding and fitting, the main geometrical unit is the sensor. On pages 8 and 9 of the report we presented at Beijing, I count 8228 sensors in the barrel silicon strip tracker. As a rough guess I will multiply this by 3 to include sensors in the forward tracker plus the sensors in the pixel system. In round numbers we have of order 25,000 sensors in the entire system.

In the following examples I will show some code fragments. For now they are written in "mock C++", which hopefully is clear enough for this point in the discussion. Also, I have paid no attention to the uses of inheritance and polymorphism; I think we need first to decide what we want and then to decide where inheritance and polymorphism improve the design.

Given a hit, we need fast, easy access to the geometry information. I propose that the way to do this is:

1. To assign each wafer a unique Id.
2. For the geometry system to support a function that returns sensor geometry information given the sensor Id.

The corresponding code fragment is,

```
class SensorId;
class SensorGeom;
SensorGeom& getSensorGeom( SensorId );
```

where `SensorGeom` is the information we need and where the function `getSensorGeom` lives within the appropriate scope within the geometry system. An illustration of the information that must be provided by `SensorGeom` is in Appendix A.

My guess is that `SensorId` will just be a dense integer, so that one can use it as an index into indirection arrays. But for now it's just a class.

I have tried to choose class names that do not match any of the existing org.lcsim or SLIC classes. The intention is to give us nomenclature to separately discuss the classes we need and the classes we have. Later we can discuss how to morph the classes we have into those that we need.

2 Local and Global Coordinate Systems

In his recent talk, Rich defined the (u, v, w) notation to describe local coordinates on a particular sensor. In this note (x, y, z) are always global coordinates and (u, v, w) are always local coordinates. As a reminder, for a strip sensor, the local coordinates are define by one vector and three unit vectors,

$\vec{r}_0$	vector to the local origin on the sensor.
$\hat{u}$	in the plane of the sensor, along the measurement direction.
$\hat{w}$	normal to the plane.
$\hat{v}$	$\hat{w} \times \hat{u}$, in the plane of the sensor, along the strip direction.

These vectors are expressed in global coorinates; that is, $\vec{r}_0$ is the location, in global coordinates, of the local origin of a sensor, and $\hat{u}$ is the measurement direction on some sensor plane, expressed in global coordinates. Note that, for an ideally aligned barrel strip sensor, $\hat{v}$ is along $\pm\hat{z}$ and, for an ideally aligned forward sensor, $\hat{w}$ is along $\pm\hat{z}$.

For a pixel sensor, $\hat{u}$ and $\hat{v}$ are the two orthogonal measurement directions in the plane.

3 Data Flow for Experimental Data I

This section discusses data flow from the detector to the tracking algorithms. It will illustrate how the geometry information can be connected to the hits in a way that I think is economical of both memory and CPU time. For simplicity I will only consider barrel strip detectors but I will presume that we want to consider some barrel layers which have two sensor layers, the second containing some sort of stereo information. The data flow is illustrated in Figure 1 and the corresponding classes are discussed below.

3.1 Packed Binary Data

The rawest form of data is packed binary data straight from the event builder. As far as this note is concerned, it is just a blob that can be unpacked. We do not need to know the internal structure. This is represented by the top box in Figure 1.

3.2 Unpacked Raw Data

The first step in processing the data is to unpack it in order to provide structured access to the information. This process should be lossless and it should report errors when it encounters inconsistent data. This produces the second box from the top in Figure 1. This box holds a single container of objects called `UnpackedSensorInfo`, each of which represents a single hit strip:

```
class ElectronicAddress{
  // Not specified in detail.
  // It contains a chip Id, a channel number within the chip ...
};

class TimeStamp{
  // Not defined in detail.
};

class UnpackedSensorInfo{

public:
  // Data members
  ElectronicAddress add;
  short int iadc;
  TimeStamp t;
};
```

By design these classes are independent of the geometry and calibration systems. They are simply a structured representation of the data at the previous level.

I am not sure that we need a timestamp but I have included one for completeness.

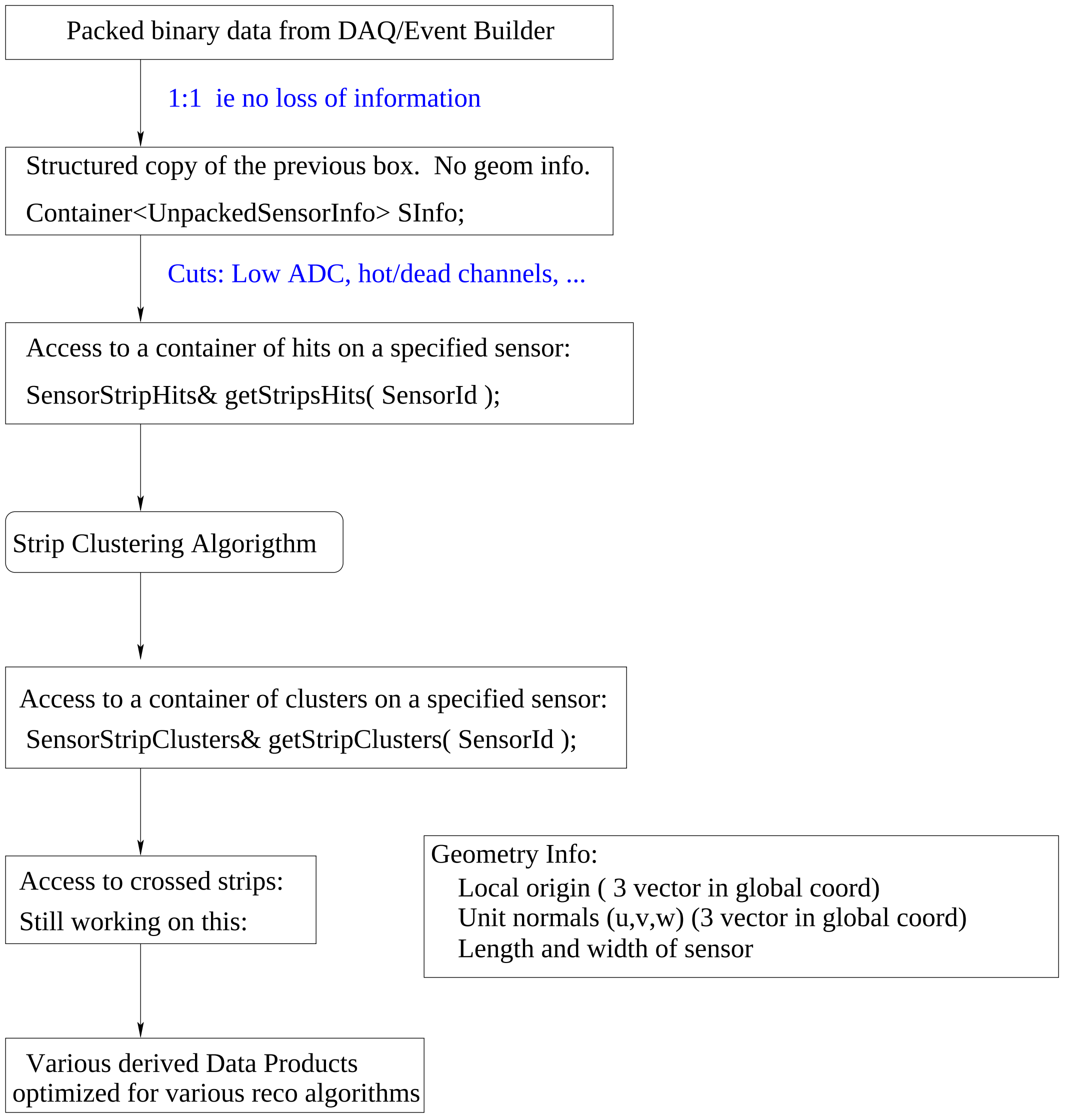
Packed binary data from DAQ/Event Builder
1:1 ie no loss of information
Structured copy of the previous box. No geom info.
Container<UnpackedSensorInfo> SInfo;
Cuts: Low ADC, hot/dead channels, ...
Access to a container of hits on a specified sensor:
SensorStripHits& getStripsHits(SensorId);
Strip Clustering Algorigthm
Access to a container of clusters on a specified sensor:
SensorStripClusters& getStripClusters(SensorId);
Access to crossed strips:
Still working on this:
Geometry Info:
Local origin (3 vector in global coord)
Unit normals (u,v,w) (3 vector in global coord)
Length and width of sensor
Various derived Data Products
optimized for various reco algorithms

Figure 1: The main objects on the path of barrel silicon strip data as it starts at the detector and is processed into tracks.

Each `UnpackedSensorInfo` object is realy just a struct whose data is complete at instantiation and never changes until the object is destructed. These objects should be treated as such; that is their only methods are their constructors and destructors. We can debate another day if we should make the data private and provide accessor functions.

3.3 Connecting the Hits to the Geometry

The next box in in Figure 1 contains the first objects in which the hit information is connected to the geometry and the conditions data. This box holds many containers of hit strips, one container per sensor. The object representing a single hit strip is,

```
typedef int StripNumber;
class StripHit{
public:
  StripNumber iu;
  short  int iadc;
  double adc;
  TimeStamp t;
};
```

These objects are collected into containers, one per sensor,

```
class SensorStripHits{
public:
  SensorId sid;
  SensorGeom& sgeom;
  Container<StripHit> hit;
};
```

where I have used "Container" as a placeholder for your favorite container type. The object that holds all of the hits for all of the sensors will look something like,

```
map<SensorId,SensorStripHits> SSHits;
```

Here I have specified the container as a map for conceptual purposes. But if the `SensorId` is just a dense integer, this can be implemented an array of SensorStripHits:

```
Container<SensorStripHits> SSHits[SensorId];
```

Whatever the case, the system can hide the implementation details if it has the following two methods for access to the hits on a given sensor:

```
bool             hasStripHits( SensorId );
SensorStripHits& getStripHits( SensorId );
```

Some additional comments on this design,

1. The class `SensorStripHits` contains a reference to the geometry of the sensor. This is the main point of this document: it is how we can connect the geometry to the hits with no copies of the geometry and the fewest reference objects.
2. This reference is computed only once per sensor per event.
3. The class `SensorStripHits` contains a copy of the SensorId. This is might be redundant but it also might be useful for debugging.

The transformation from the previous box, in its full complexity, is a heavy-weight step:

1. It needs access to the full conditions data base to access pedestals and gains for the ADCs, to access the list of hot/dead channels, and to access the map from electronic addresses to sensor/strip.
2. It is allowed to discard data. One might discard data because the ADC is below threshold, because the strip is on a list of known hot/dead channels or because the hit is out of time.
3. There no links from this container back to precursor containers: the only repeated data is `iadc`, which is lighter weight than a back link.

A Information Needed in `SensorGeom`

The following class illustrates the information needed from the geometry system for each sensor.

```
class SensorGeom{

public:
  SensorID id;
  SubSystem type;          // See below.  Might not be needed?
  double r[3];             // $\hat{r}_0$.
  double u[3], v[3], w[3]; // unit vectors along the u,v,w axes.
  double dim[3];           // Dimensions along the u,v,w axes.
  double pitch[2];         // Pitch along the u and v axes.

                           // Do we need?
  ?????                    // A reference back to full geom info?
};
```

where SubSystem is a bookkeeping convenience defined by,

```
enum SubSystem { BarrelPixel, ForwardPixel,
                 BarrelStrip, ForwardStrip};
int nSubSystems(ForwardStrip+1);
```

If the equivalent information is already directly, *and quickly*, available, from the existing geometry system, we do not need this class and our code can talk directly to the existing geometry system.

It is redudant to carry the SensorId around in the object but I find it a powerful debugging convenience.

I have not defined any methods for this class since it is essentially a convenience class for accessing a subset of the information on each sensor. If someone wants the full power of the geometry system, they can follow the pointer/reference back to the geometry system and access its methods directly.

To complete the geometry specification we need to define a few conventions, such as: Conventions we need to define to complete this:

- Where is the local orign: at the center of the "top" surface, at the body center, at a corner, at an edge?
- Are dimension full lengths or half lengths?
- Is $\hat{w}$ an inward normal or outward normal or someother convention.
- For pixels, which is u and which is v?
- Do we want to define now the methods one would need to deal with non-flat wafers?